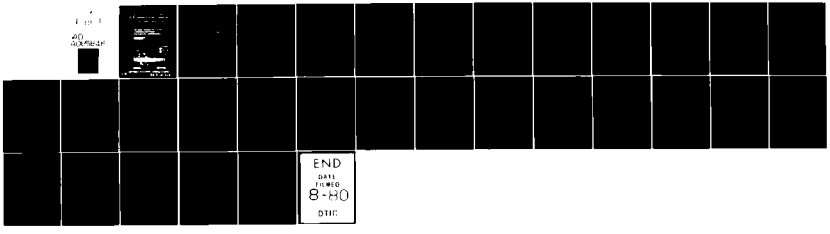
AD-A085 846 ILLINOIS UNIV AT URBANA-CHAMPAIGN APPLIED COMPUTATION--ETC F/G 9/5
THE CUBE-CONNECTED-CYCLES: A VERSATILE NETWORK FOR PARALLEL COM--ETC(U)
NOV 79 F P PREPARATA, J VUILLEMIN N00014-79-C-0424
UNCLASSIFIED ACT-20 NL



END
DATE FILMED
8-80
DTIC

ACT-20

LEVEL II

12

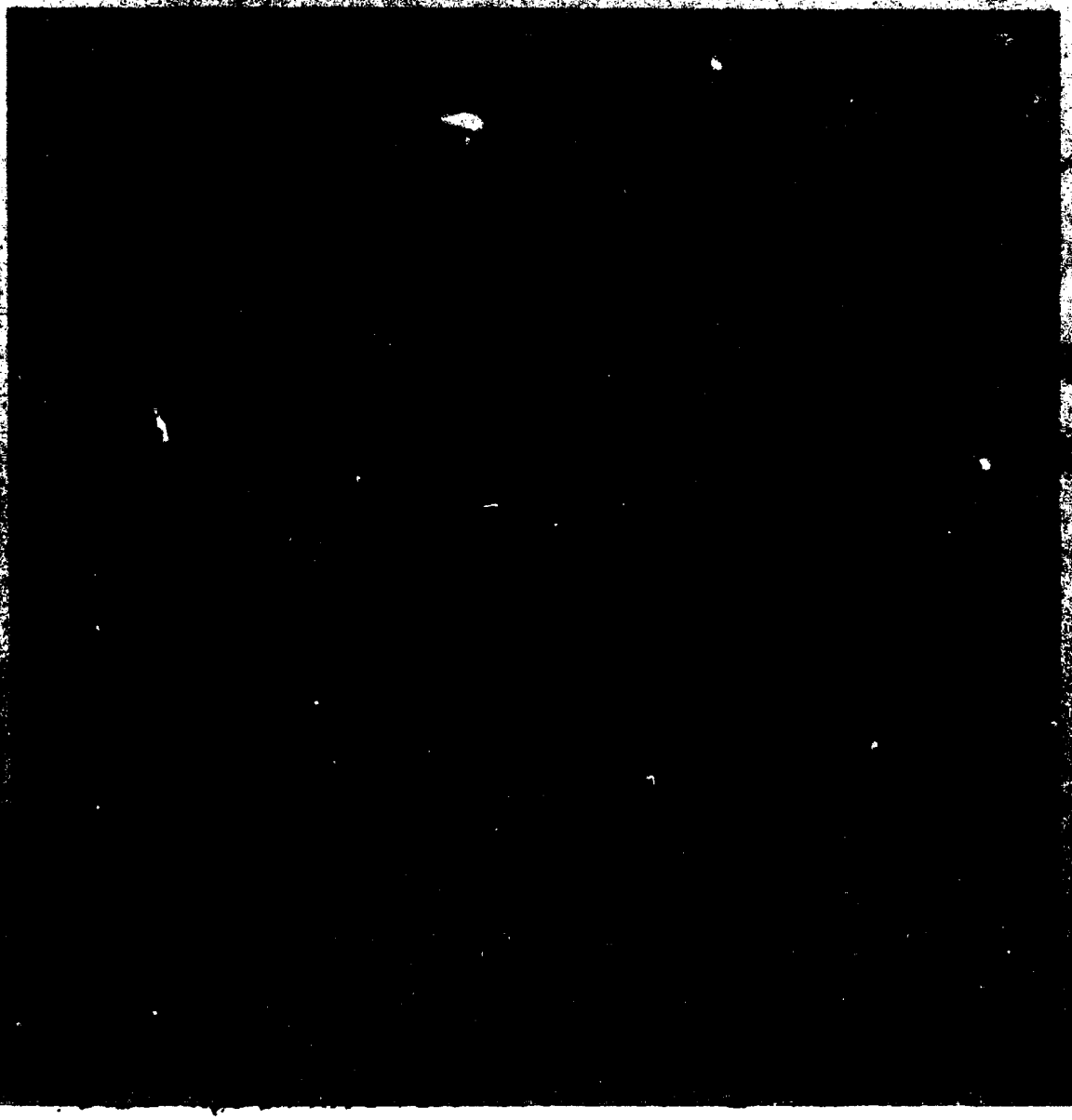
NOVEMBER, 1979

CSL COORDINATED SCIENCE LABORATORY

APPLIED COMPUTATION THEORY GROUP

ADA085846

THE CUBE-CONNECTED-CYCLES: A VERSATILE NETWORK FOR PARALLEL COMPUTATION



SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
2. GOVT ACCESSION NO. AD-A085 846
3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
(1) THE CUBE-CONNECTED-CYCLES: A VERSATILE NETWORK FOR PARALLEL COMPUTATION

5. TYPE OF REPORT & PERIOD COVERED
(9) Technical Report

6. PERFORMING ORG. REPORT NUMBER
R-874(ACT-20);UILU-ENG80-2206

7. AUTHOR(s)
(10) Franco P. Preparata
Jean Vuillemin

8. CONTRACT OR GRANT NUMBER(s)
NSF MCS-78-13642; N00014-79-C-0424

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
12

11. CONTROLLING OFFICE NAME AND ADDRESS
(11) National Science Foundation;
Joint Services Electronics Program Contract

12. REPORT DATE
Nov[redacted]79

13. NUMBER OF PAGES
27

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)
(14) ACT-20, UILU-ENG 80-2206

15. SECURITY CLASS. (of this report)
UNCLASSIFIED

15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

(15) N00014-79-C-0424, NSF-MCS 78-13642,

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Parallel processing, VLSI design, sorting, Fourier Transform

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We introduce an interconnection pattern of processing elements, the cube-connected-cycles (CCC), which can be used as a general purpose parallel processor. Because its design complies with present technological constraints, the CCC can also be used in the layout of many specialized large scale integrated circuits (VLSI). By combining the principles of parallelism and pipelining, the CCC can emulate the cube-connected machine and the perfect shuffle with no significant degradation of performance but with a more compact structure. We describe in detail how to program the CCC for efficiently

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

410809

(over)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. (Continued)

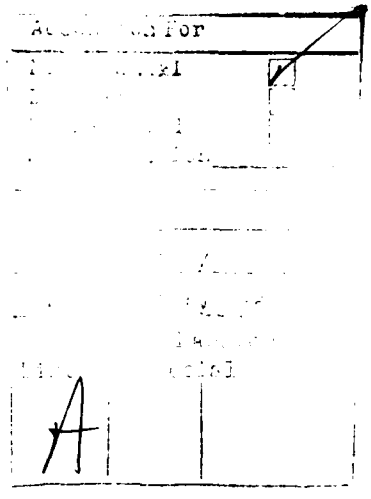
solving a large class of problems, which includes Fast-Fourier-Transform, sorting, permutations, and derived algorithms.

THE CUBE-CONNECTED-CYCLES: A VERSATILE NETWORK FOR PARALLEL COMPUTATION

by

Franco P. Preparata and Jean Vuillemin

This work was supported in part by the National Science Foundation under Grant MCS 78-13642 and Joint Services Electronics Program under Contract N00014-79-C-0424.



A preliminary version of this report was issued in June 1979 by I.R.I.A., Institut de Recherche d'Informatique et d'Automatique, 78150 Rocquencourt, France.

THE CUBE-CONNECTED-CYCLES: A VERSATILE NETWORK FOR PARALLEL COMPUTATION

Franco P. Preparata

Coordinated Science Laboratory
University of Illinois
Urbana, Illinois 61801

Jean Vuillemin

Laboratoire de Recherche en Informatique
Université de Paris-Sud
91405 Orsay, France

Abstract: We introduce an interconnection pattern of processing elements, the cube-connected-cycles (CCC), which can be used as a general purpose parallel processor. Because its design complies with present technological constraints, the CCC can also be used in the layout of many specialized large scale integrated circuits (VLSI). By combining the principles of parallelism and pipelining, the CCC can emulate the cube-connected machine and the perfect shuffle with no significant degradation of performance but with a more compact structure. We describe in detail how to program the CCC for efficiently solving a large class of problems, which includes Fast-Fourier-Transform, sorting, permutations, and derived algorithms.

Keywords: Parallel processing, VLSI design, sorting, Fourier Transform.

CR Categories:

† This work was partially supported by National Science Foundation Grant MCS-78-13642, by the Joint Services Electronics Program Contract DAAG-29-78-C-0016, by I. R. I. A., Institut de Recherche en Informatique et Automatique, 78150 Le Chesnay, France, and by ERA 452 "al Khowarizmi" of Centre National de la Recherche Scientifique.

‡ A preliminary version of this paper has been presented at the 20th Annual Symposium on Foundations of Computer Science, Puerto Rico, Oct. 1979.

1. INTRODUCTION

The great technological progress embodied in very large scale integration (VLSI) of electronic circuits has made it possible to conceive large systems of processing elements cooperating in the execution of parallel algorithms. This has motivated considerable research interest in parallel computation. Unfortunately, here the situation is very different from that of serial computation, where the RAM machine [1] represents a universally accepted model. The difficulty of choosing a specific interconnection is frequently bypassed by assuming a model (shared-memory-machine) where each pair of processors is connected (or an equivalent system) [2-5]. Although not without merit, because it aims at uncovering the inherent data-dependence of given problems, this approach ignores the technological constraints of VLSI, particularly as regards the communication among the processing elements [6]. At the opposite end, other workers [7-11] suggest that processor interconnection should be limited to planar links between topologically neighboring cells (arrays or meshes). Such designs are certainly well suited for current VLSI technology, and they have cleverly been used in implementing algorithms for matrices or graph problems [9-12], for example. This type of connection, however, is not suited for efficiently implementing algorithms for various fundamental problems, such as sorting and convolution. Indeed, good algorithms for solving these problems intrinsically require data movement between processors which are topologically far apart; for example, sorting on an n processor array such as ILLIAC IV requires time $\Omega(\sqrt{n})$ [8].

The purpose of the paper is to propose and analyze a new interconnection of processors, called the cube-connected-cycles, which is remarkably suited for implementing efficient algorithms such as Fast-Fourier-Transform (FFT), sorting, etc... . The geometric structure underlying the interconnections is that the k-dimensional cube. This structure which has already been studied in relation to parallel computation [13], is not readily usable for VLSI design, since each of the 2^k processors is connected to k other processors.

By combining parallelism and pipelining we are able to achieve the following results:

(1) The number of connections per processor is reduced to 3.

(2) Processing time is not significantly increased with respect to that achievable on the k-cube structure.

(3) Programs for the individual modules are obtained in a systematic way from a standard description of the global algorithms.

(4) The overall structure complies with the basic requirements of VLSI technology: modularity, ease of layout, simplicity of communication among the processing elements, simplicity in timing and control of the entire system [14]. We also propose a wire layout of the CCC, which can be physically realized with two orthogonal layers of wires. This layout is optimal for several problems, according to a recently proposed VSLI model [18].

(5) Finally we are able, without resorting to any drastic departure from classical algol-like languages, to provide fully accurate and hopefully easily understandable descriptions of our parallel programs. This is a favorable sign that parallel processing may possibly be endowed with suitable high level programming languages.

This paper is organized as follows. Section 2 introduces a class of algorithms comprising many important applications, such as merging, sorting, Fourier Transform, data rearrangement, ... Section 3 presents models of module connections, including the CCC, allowing for efficient parallel execution of the algorithms in Section 2. Section 4 describes the implementation of such algorithms on the CCC, and Section 5 is devoted to optimality considerations regarding a layout of the machine for VLSI realizations.

2. A CLASS OF HIGHLY PARALLEL ALGORITHMS

To describe our algorithms, assume that input data $t_0, t_1, \ldots, t_{n-1}$ are stored respectively in storage locations T[0],T[1],...,T[n-1], and that $n = 2^k$, i.e., the number of inputs is a power of 2. We say that an algorithm is in the DESCEND class if it performs a sequence of basic operations on data which are successively $2^{k-1}, \ldots, 2^j, \ldots, 2^0 = 1$ locations apart. Each basic operation OPER(m,j;U,V) modifies the two data items present in storage locations U and V; the computation performed affects only the contents of U,V and it may depend upon parameters m and j, which are integers $0 \leq m < n$, $0 \leq j < k$.

Algorithms in the DESCEND class are then specified as:

```
proc  DESCEND
      for j ← k-1 step -1 until j = 0
      do foreach m: 0 ≤ m < n
         pardo if bit_j(m) = 0 then OPER(m,j;T[m],T[m+2^j])
               fi
         odpar
      od
corp  DESCEND .
```

Here, $\text{bit}_j(m)$ is the coefficient of 2^j in the binary representation of $m = \sum_{j \geq 0} \text{bit}_j(m) 2^j$. The language construct **foreach** m: <cond(m)> **pardo** <action> **odpar** obviously indicates that all instructions <action> corresponding to values of m satisfying <cond(m)> can be performed simultaneously. On machines where such parallelism can be realized, DESCEND algorithms run in $k = \log_2(n)$ elementary steps.

We also introduce the dual class ASCEND, where the control of the algorithm is changed to

for j ← 0 **step** 1 **until** j = k-1,

i.e., OPER is performed on data which are successively $1 = 2^0, 2^1, \ldots, 2^j, \ldots, 2^{k-1}$ locations apart. To clarify the duality between ASCEND and DESCEND consider the binary representation of $m = \sum_{0 \le i < k} bit_i(m) \cdot 2^i$ and define $\tilde{m} = \sum_{0 \le i < k} bit_i(m) \cdot 2^{k-i-1}$, the integer whose binary representation is the reversal of that of m. Once k is fixed, the function: $m \to \tilde{m}$ is an involutory permutation of $0, 1, \ldots, 2^k - 1$ known as the bit reversal permutation (BRP). For example, for k = 3, the BRP of (0 1 2 3 4 5 6 7) is (0 4 2 6 1 5 3 7).

By first applying the BRP to its inputs, an ASCEND algorithm can be transformed into a dual DESCEND algorithm (figure 1) whose basic operation $\widetilde{OPER}$ is related to the original OPER by:

$$\widetilde{OPER}(m,j;U,V) = OPER(\tilde{m},k-1-j;U,V)$$

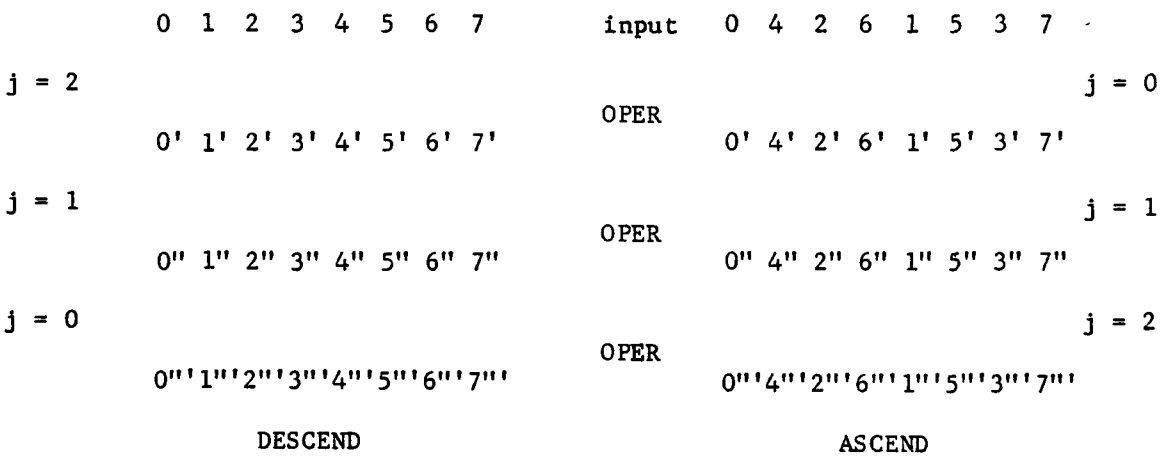


Figure 1. Dual algorithms; operands are denoted by their original addresses, connecting lines show interacting operands, and priming indicates the number of operations through which an operand has been processed.

It is now time to exhibit algorithms for solving specific interesting problems. Some applications - such as bitonic merge and cyclic shift - are directly within the ASCEND or DESCEND classes (simple algorithms); for these applications, all we have to do is specify OPER(m,j;U,V).

Other applications (such as permutation, shuffle, unshuffle, bit-reversal (BRP), odd-even-merge, Fast-Fourier-Transform, convolution, matrix transposition) have programs consisting of a short sequence of algorithms (cascaded algorithms) in the preceding class, and thus run in O(logn) parallel steps.

We also have applications - such as bitonic sort, odd-even-sort, and calculations of symmetric functions - for which the combining step of the two results of a recursive call is itself an algorithm in one of the two preceding categories. These algorithms, which we call composite, run in $O((\log n)^2)$ parallel steps.

2.1 Bitonic Merge

The elegant algorithm for bitonic merge, due to K. E. Batcher [15], is ideally suited for implementation within the DESCEND class. All we need is to specify OPER(m,j;U,V) as a comparison-exchange. Precisely, in order to handle sequences which are sorted either in increasing or in decreasing order, we define ORIENTCOMPEXCHANGE(m,j;U,V) as

```
if bit_j(m) = 0 then (U,V)←(min (U,V), max (U,V))
                else (U,V)←(max (U,V), min (U,V))
fi .
```

Batcher's odd-even merge [15,16] can also be programmed as a cascaded algorithm, running in O(logn) parallel steps.

2.2 Radix-2 Fast-Fourier-Transforms and Convolution

The important FFT algorithm can be set in the ASCEND class. Let ω be a primitive root of unity of order $n = 2^k$. If $\langle A_0,\ldots,A_{n-1}\rangle$ is the Fourier Transform of vector $\langle a_0,\ldots,a_{n-1}\rangle$, it is well-known that $A_j = U_j + \omega^j V_j$ and $A_{j+2^{k-1}} = U_j - \omega^j V_j$ where the U's and V's

are respectively the Fourier Transforms, with primitive root ω^2, of the "even" subsequence $\langle a_0, a_2, \ldots, a_{2^k-2} \rangle$ and the "odd" subsequence $\langle a_1, a_3, \ldots, a_{2^k-1} \rangle$; we call the ω^j's the combining root powers.

The above relationships indicate that the sequence $\langle a_0, \ldots, a_{n-1} \rangle$ must be initially rearranged by means of the bit-reversal permutation. Once the desired reconfiguration has been achieved, we may proceed with the actual FFT computation, which is in the ASCEND class.

Its basic operation OPER(m,j;U,V) is specified by

$(U,V) \leftarrow (U+\alpha V, U-\alpha V)$ where $\alpha = \omega^{m \cdot 2^{k-j}}$.

It is not hard to show that α can be computed efficiently at each step; precisely, the time used by each module to compute, by successive squaring, the required combining root powers for the entire algorithm is $O((\log\log n)^2) = o(\log n)$. Using a sequence of two inverse Fourier transforms in the classical manner [1] allows one to compute the convolution of two sequences, from which a wealth of applications can be derived (see [1]).

2.3 Data Rearrangements

Being able to efficiently permute the data is obviously important for may applications. For example, the BRP rearrangement is a necessary preliminary step to the FFT algorithm of the preceding section. Some permutations, such as cyclic shifts, shuffle, and unshuffle can be computed by algorithms in ASCEND or DESCEND, as the reader will enjoy discovering for himself (here "shuffle" of $(0,1,2,\ldots,2^k-1)$ is $(0,2^{k-1},1,2^{k-1}+1,\ldots,2^{k-1}-1,2^k-1)$ and "unshuffle" is the inverse permutation). Other permutations, such as BRP or matrix transpose, are computed by cascaded algorithms. In general, we can emulate a Benes permutation network [21] by a sequence ASCEND;DESCEND, thus in time $O(\log n)$; it must be pointed out, however, that to realize an arbitrary permutation, the exchange information must be precomputed.

2.4 Sorting and Calculation of Symmetric Functions

The previously described merge routines can be used as the basis of efficient sorting algorithms. A sequence of input keys is divided into two halves, each of which is recursively sorted (in opposite order in the case of bitonic sort), and then merged using either of the above merge routines. Both algorithms run in time $O((\log n)^2)$.

One can compute symmetric functions in a completely analogous fashion: apply recursive calls to each half of the data, and compute the convolution of the two resulting sequences, again in time $O((\log n)^2)$.

2.5 Matrix Multiplications and Other Algorithms

To compute the matrix product $C = A \times B$ of two $n \times n$ matrices, we must obviously first store $A = (A_0^T \ldots A_{n-1}^T)^T$ in row major order, and $B = (B_0 \ldots B_{n-1})$ in column major order. Assuming we have enough space and processors, i.e., $2^k \geq n^3$, we copy A and B into the pattern:

$A_0B_0A_0B_1 \ldots A_0B_{n-1}A_1B_0 \ldots A_iB_j \ldots A_{n-1}B_{n-1}$. All this can be achieved with simple-minded cascaded algorithms, in time $O(\log n)$.
Each of the scalar products $c_{i,j} = A_i \cdot B_j = \Sigma\, a_{i,k} \cdot b_{k,j}$ is computed in parallel, within $O(\log n)$ additional time units. The results $c_{i,j}$ are then regrouped, according to the output format (say, row major).
Although the details of this algorithm are a bit tedious to describe, it should be clear that <u>matrix multiplication can be computed in time $O(\log n)$</u>, within our class of algorithms. In fact, a surprising number of other algorithms can be efficiently implemented within this framework, including all of the interesting algorithms for parallel processing known to the authors.

3. DESCRIPTION OF THE SCHEME

In order to efficiently implement algorithms in the DESCEND class, the most natural interconnection of modules is that of the k-dimensional binary cube (k-cube) where each of the 2^k processors is numbered from 0 to 2^k-1 and is connected to each of the k processors whose binary numbering differs in exactly one binary position (figure 2). Although an ASCEND or DESCEND algorithm can be implemented on such a machine in $\log_2 n$ parallel steps, this proposal is not feasible mainly because the number $k = \log_2 n$ of connections for each processor is too large. The unfolded k-cube and the perfect shuffle interconnections have been proposed [17] (figure 3), as attempts to remedy this difficulty.

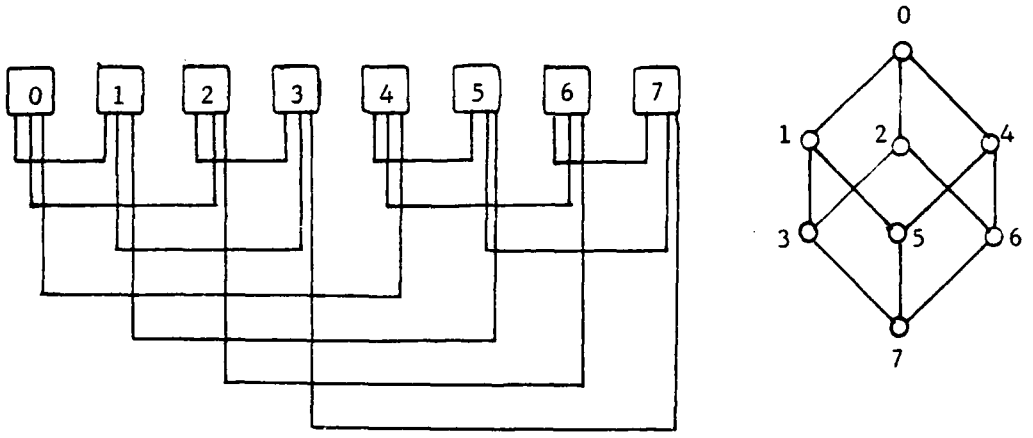


Figure 2. The 3-cube.

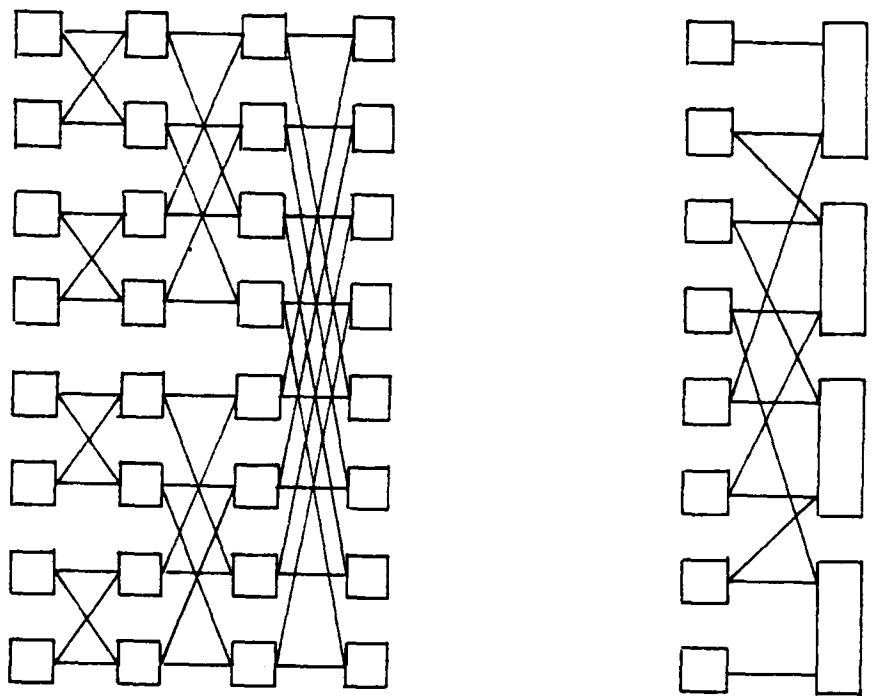


Figure 3. Unfolded 3-cube (left) and perfect shuffle (right) interconnections.

Although both structures have a fixed number (4) of connections per processor, their intrinsic topology make them inferior, as regards physical layout (see section 5), to the scheme we now describe.

Our parallel computing system, the cube-connected-cycles (CCC), is a network of identical processors, called modules. A module has 3 interconnection ports. Each interconnection line linking two modules can be used for the bidirectional transmission of one operand, and it is irrelevant here whether operand transmission is serial or parallel. For correctly executing the algorithms described in the preceding sections, it is indifferent to synchronize the entire system through a central clock, which defines time units for all modules, or to let synchronization

problems be settled at the level of each communication line, thus achieving a globally asynchronous system. In order to describe the interconnections, we assume for simplicity that n, the number of modules, is a power of two, i.e., $n = 2^k$, and, moreover, assume that k is of the form $k = r + 2^r$; the modifications resulting when k is arbitrary are straightforward (in the latter case, r is the smallest integer for which $r + 2^r \geq k$). Each module has a k-bit address m which in turn is expressed as a pair (ℓ,p) of integers represented with (k-r) and r bits respectively, such that $\ell \cdot 2^r + p = m$.

As mentioned earlier, each module has three ports: F, B, and L (mnemonic for forward, backward, lateral), whose connection is entirely determined by the module address (ℓ,p), that is:

$F(\ell,p)$ is connected to $B(\ell,(p+1) \bmod 2^r)$

$B(\ell,p)$ is connected to $F(\ell,(p-1) \bmod 2^r)$

$L(\ell,p)$ is connected to $L(\ell + \epsilon\ 2^p,p)$

where $\epsilon = 1\text{-}2\text{bit}_p(\ell)$. The interconnection scheme is displayed in figure 4. In words, the modules are grouped into 2^{k-r} cycles, each cycle consisting of 2^r modules, cyclically connected by the F-B lines. The cycles are in turn interconnected as a (k-r)-cube; if $\langle x_0,x_1,\ldots,x_{k-r-1}\rangle$ are the dimensions of the (k-r)-cube, all edges along dimension x_i, called collectively sheaf i, link modules whose addresses are (.,i). The total number of interconnection links is $3 \cdot 2^{k-1} = \frac{3}{2} \cdot n$.

Each module contains an operand register T, a few memory locations, and possesses basic arithmetic and logical capabilities. It is controlled by a stored program or a circuit implementation of such a program. For the time being, we make the hypothesis of unlimited parallelism,

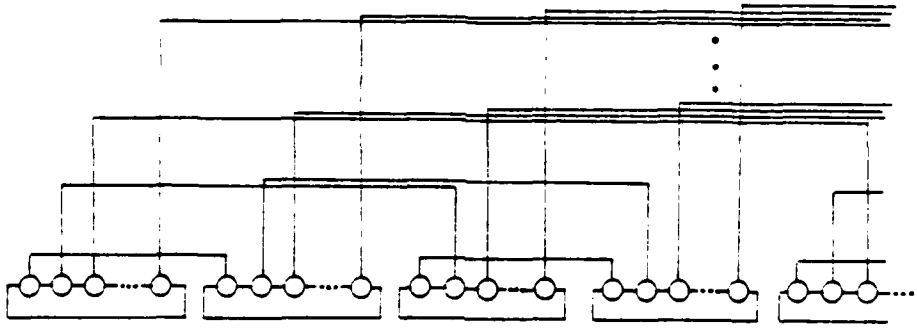


Figure 4. The CCC interconnection scheme.

that is, the number of modules is tailored to the problem size; under this hypothesis, the one or two memories mentioned earlier suffice. Subsequently (section 4.3), under the hypothesis of limited parallelism, we shall endow each module with a small private random access memory. In either case, each module is somewhat simpler than a current microprocessor but not basically different from it.

4. EMULATION OF THE k-CUBE ON THE CCC

In order to implement DESCEND on the CCC, we prune the k-cube so as to use only connections existing in the CCC. The first stage consists in removing the sheaves corresponding to dimensions 0,1,...,r-1, and using instead the cycle connections F and B, as introduced in section 3. Our original DESCEND program is thus transformed to:

```
proc  DESCEND
    for j ← k-1 step-1 until j = r
    do foreach m: 0 ≤ m < n
       pardo if bit_j(m) = 0 then OPER(m,j;T[m],T[m+2^j])
            fi
       odpar
    od;
    foreach ℓ: 0 ≤ ℓ < 2^(k-r) pardo LOOPOPER(ℓ) odpar
corp  DESCEND.
```

Here procedure DLOOPOPER(ℓ) processes the data within cycle ℓ to compute the desired result in $O(2^r)$ parallel steps, as we show later. Note that the running time is still $O(k-r) + O(2^r) = O(\log n)$.

The second transformation consists in removing, for all $j = 0,\ldots,k-r-1$, the k-cube links pertaining to sheaf $(r + j)$, except those existing between modules whose addresses are of the form $(.,j)$: the resulting interconnection is then exactly the one of the CCC, as introduced in Section 3.

The computation corresponding to the *for* loop of the above algorithm can no longer be performed in one parallel step. Using repeated circular shifts within cycles, however, each operand in the

cycle can be successively brought to reside for one time unit in module (.,j), where OPER(.,j;.,.) can then be executed. Although the execution of OPER(.,j;.,.) for all operands in a cycle now requires 2^r time units, this computation can be pipelined (overlapped) with the analogous operations OPER(., i;.,.) for $r \leq i < k$. To achieve pipelining thus requires a new function BSHIFT(ℓ), which performs a cyclic backward shift of the operands in cycle ℓ, that is:

foreach j: $0 \leq j < 2^r$ pardo $T[\ell \cdot 2^r + ((j-1) \bmod 2^r)] \leftarrow T[\ell \cdot 2^r + j]$

odpar.

The final version of DESCEND is thus:

```
proc   DESCEND
     for i ← 2^r-1 step-1 until i = -2^r
     do foreach ℓ: 0 ≤ ℓ < 2^(k-r)
        pardo foreach p:max(i,0) ≤ p < min(2^r,2^r+i)
              pardo if bit_p(ℓ) = 0 then OPER(a,b;U,V)
                    where a = ℓ·2^r+((p+i-1)mod 2^r),
                          b = p+r,
                          U = T[ℓ·2^r+p],
                          V = T[(ℓ+2^p)·2^r+p].
                    fi
              odpar;
              BSHIFT(ℓ)  Comment  backwards shift of cycle ℓ;
     od;
     Comment end of treatment on sheaves k-1,k-2,...,r;
     foreach ℓ: 0 ≤ ℓ < 2^(k-r) pardo LOOPOPER(ℓ) odpar
corp  DESCEND.
```

The inner operation of the <u>for</u> loop is executed in two time units; one for OPER, then one for BSHIFT. The total running time is thus $4 \cdot 2^r$ plus the time for executing LOOPOPER. If we can ensure that LOOPOPER can be processed in time linear in the cycle size, the entire procedure will be executed on the CCC in time O(logn).

Figure 5 provides a schematic view of DESCEND on the CCC, and conventions used are those of figure 1, which depicts DESCEND on the k-cube. Here we assume k = 3, thus the CCC consists of 4 cycles of length 2.

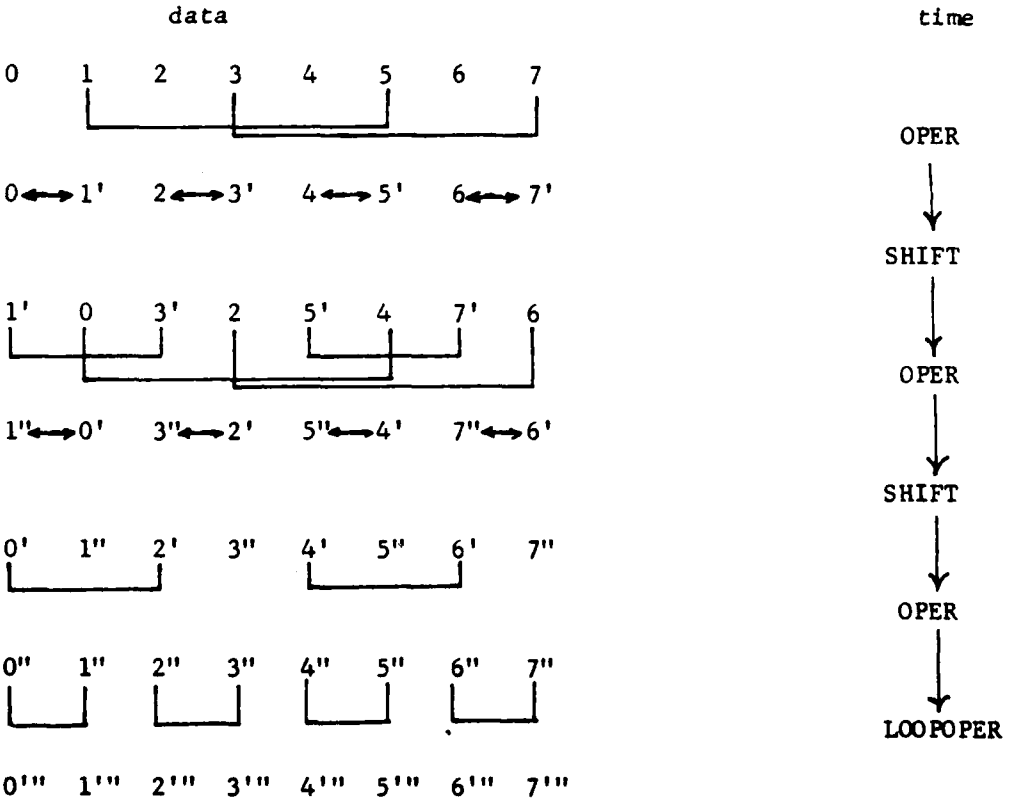


Figure 5. DESCEND on the CCC, k = 3

4.1 Computation Within the Cycles

The next question to be addressed is the implementation of LOOPOPER(ℓ), so that it runs in time linear in the cycle length. Obviously, we are constrained to using only the F and B cycle links existing in the CCC. Our objective is to emulate, on the cycle of length 2^r, the operation OPER as it would be executed on hypothetical r-cube sheaves. Since OPER may take place in the cycle only between adjacent modules, particular care must be exercised to ensure that the desired adjacencies, corresponding to all sheaves, be globally realized in time linear in the cycle length. The key permutations for this task are based on the perfect unshuffle [16,17]. Specifically, UNSHUFFLE(ℓ,i) performs the perfect-unshuffle operation on each of the 2^{r-i-1} contiguous blocks of length 2^{i+1} into which $T[\ell \cdot 2^r :: (\ell+1) \cdot 2^r - 1]$ is subdivided, and is realized as follows:

```
proc  UNSHUFFLE(ℓ,i)
      for b ← 2^i step-1 until b = 2
      do foreach m: m = ℓ·2^r + (2·s+1)·2^i + p
                  where 0 ≤ s < 2^(r-i-1), -b < p < b,
                        (p mod 2) = (b mod 2)
                  pardo T[m-1] ↔ T[m] odpar
      od
corp UNSHUFFLE.
```

Clearly, UNSHUFFLE(ℓ,i) runs in (2^i-1) parallel step. It is also easy to realize that the program

```
proc BRP(ℓ)
   for i ← r-1 step-1 until i = 1 do UNSHUFFLE(ℓ,i) od
corp BRP
```

realizes the bit-reversal permutation of $T[\ell \cdot 2^r :: (\ell+1)2^r - 1]$ with reference to the r least-significant bits of the addresses.

We can now elucidate the general format of LOOPOPER, which consists of a sequence of unshuffle-operation pairs, each emulating a sheaf operation. This is preceded by BRP, so that, upon completion, the results are in the correct order (see figure 6). In the description below the parameter a gives the original address of the operand which is brought to module (ℓ,p) by the sequence: BRP; UNSHUFFLE$(\ell,0)$; UNSHUFFLE$(\ell,1)$;...;UNSHUFFLE$(\ell,r-1-j)$. (Recall that $\tilde{q}$ denotes the integer whose binary representation is the reversal of that of the integer q.)

```
proc  LOOPOPER(ℓ)
      BRP(ℓ);
      for j ← r-1 step-1 until j = 0
      do foreach q: 0 ≤ q < 2^r, bit_0(q) = 0
            pardo OPER(a,j;T[ℓ•2^r+q],T[ℓ•2^r+q+1])
                  where a = ℓ•2^r+(q̃ mod 2^j) + (q mod 2^(r-j))•2^j.
            odpar;
        UNSHUFFLE(ℓ,j)
      od
corp LOOPOPER.
```

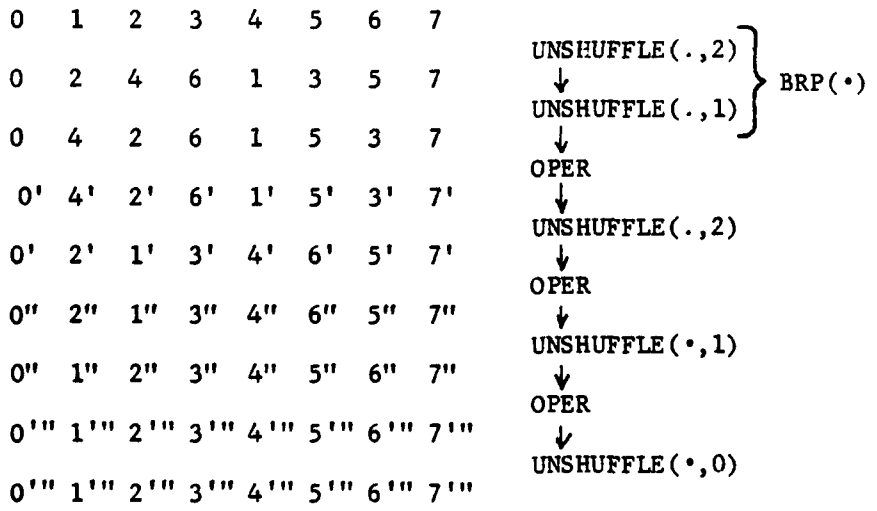


Figure 6. A schematic presentation of LOOPOPER for r = 3.

With respect to execution time, we noted that UNSHUFFLE(•,i) runs in time $O(2^i)$; thus BRP and LOOPOPER jointly run in $O(1+2+2^2+\ldots+2^{r-1}) = O(2^r)$ steps, linear in the cycle length.

4.2 Programs for each Module of the CCC

From the preceding global description of DESCEND, it is rather straightforward to produce the sequential program of module (ℓ,p). The program MODULE(ℓ,p) for a given DESCEND algorithm is of the form: HIGHSHEAVES(ℓ,p);LOWSHEAVES(ℓ,p), which respectively implement the (k-r)-cube operation and LOOPOPER. The entire MODULE(ℓ,p) is of a very simple nature: it basically counts up time and, at each time unit numbered t, it tests a simple logical condition involving ℓ,p, and t; depending on this test, either it does nothing, or it exchanges operands, or it exchanges operands and performs an operation on them. The details of these programs are omitted for the sake of brevity.

The precise execution time of DESCEND (or ASCEND) on the CCC is given by the formula:

$$T = 4.2^r \cdot T_{CCC} + (r+2^r)T_{oper}$$

where T_{CCC} is the time required for stepping up the control variable t, testing it and performing one data exchange on some of the links; T_{oper} is the time required for computing OPER(m,j;U,V) within each module.

4.3 Limited Parallelism

So far, we have assumed that the size n of the CCC was tailored to the application. To cope with the realistic situation where the number N of inputs is larger than the size n of the CCC, we suggest to let each module of the CCC be a full fledged microprocessor endowed with a private RAM memory.

Assuming for simplicity that $N = sn$, with $s = 2^q$ integer, we require that the RAM memory of each module be of size s and denote by $T[m,0::2^q-1]$ the private memory locations of module m. The input $a_0,\ldots,a_{N-1}$ is divided into consecutive blocks of size s, each block being stored within a module of the CCC, so that $T[m,j] = a_{2^q \cdot m+j}$ for $0 \leq j < 2^q$.

The only modification concerns the program MODULE(ℓ,p) (see Section 4.2), which now assumes the format HIGHSHEAVES(ℓ,p);LOWSHEAVES(ℓ,p); LOCAL(ℓ,p). Programs for HIGHSHEAVES and LOWSHEAVES are the same as before, except that each operation and data transmission is now successively performed on the 2^q data items of each module. As for LOCAL:

```
proc LOCAL(ℓ,p)
     u ← m•2^q
     for j ← q-1 step-1 until j = 0
     do for i ← 0 step 1 until i = 2^q-1
        do if bit_j(i) = 0
              then OPER(u+i,j;T[m,i],T[m,i+2^j]) fi
        od
     od
corp LOCAL.
```

It should be clear by now that all of the algorithms described in Section 1 can be applied here. A direct analysis shows that, on a CCC consisting of n processors, each processor having memory $\frac{N}{n}$, we can process N inputs in time $O(\frac{N}{n} \cdot \log N)$ for algorithms in the classes ASCEND or DESCEND, thus achieving the optimal speed-up possible with n processors.

5. LAYOUT OF THE CCC FOR VLSI

It is interesting to examine the just described CCC within the framework of the "VLSI model of computation" recently proposed [14,18,19]. In this model, each wire has unit width on the silicon chip and transmits a unit of information in a unit of time; information is taken from, or delivered to, special areas on the chip, called nexuses, each associated with a module. Within this model, which takes realistic account of the placement of modules and interconnection, C. D. Thompson has studied the implementation of the Fast-Fourier-Transform [18] and has elucidated significant relationships between input size n, chip area A, processing time T, and the so-called minimal bisection width ω.[1] Thompson has shown that $A \geq \omega^2/4$ in general, and that, for the n-point FFT, $T \geq n/2\omega$, thus establishing the bound $AT^2 \geq n^2/16$. The lower bound for time applies to a wider class of problems, as shown by the following proposition which we state without proof:

Proposition: In the VLSI model (Thompson [18]), time $T \geq \frac{n}{2\omega}$ is required to merge two sorted sequences of length n/2, or to realize the data rearrangement specified by some permutation drawn from a transitive group of permutations.[2]

As a consequence, we have $AT^2 \geq \frac{n^2}{16}$ for all such problems.

---

[1] For a graph $G = (V,E)$ the minimal bisection width ω is defined as the smallest integer such that $\omega = |\{(u,v) \in E : u \in V_1, v \in V_2\}|$, where $\{V_1,V_2\}$ is a partition of V with $|V_1| \leq |V_2| \leq |V_1| + 1$.

[2] A subgroup G of the symmetric group S_n is said to be transitive if $\forall i,j\ 1 \leq i,j \leq n, \exists \sigma \in G : \sigma(i) = j$, meaning that data located in any position of the machine may be moved into any other position of the machine.

With the CCC, we have shown that operations such as FFT, merging, cyclic shifts, shuffles, etc., are all realizable in the minimal achievable time $T = O(\log n)$. We now demonstrate that $A = O(n^2/\log^2 n)$ thus achieving the lower bound exactly; this means that the CCC is optimal in the VLSI model for FFT, merging of sorted sequences, and realization of permutations drawn from a transitive group. In contrast, known layouts for the k-cube or the perfect shuffle have area of a larger order.

To achieve $A = O((n/\log n)^2)$ for the CCC, consider a layout which uses two sheaves of evenly spaced wires, horizontal and vertical, used respectively for cube and cycle connections. Figure 7 pictorially provides base, inductive hypothesis, and extension, to prove that an $n = s \cdot 2^s$ module CCC can be placed on a $2^s \times (2.2^s - 1)$ chip; since $s \simeq \log_2(n/\log_2 n)$, the chip size is about $(n/\log_2 n) \times (2n/\log_2 n - 1) = O((n/\log n)^2)$. Slightly more complicated constructions yield somewhat more efficient module placements as suggested by figure 8.

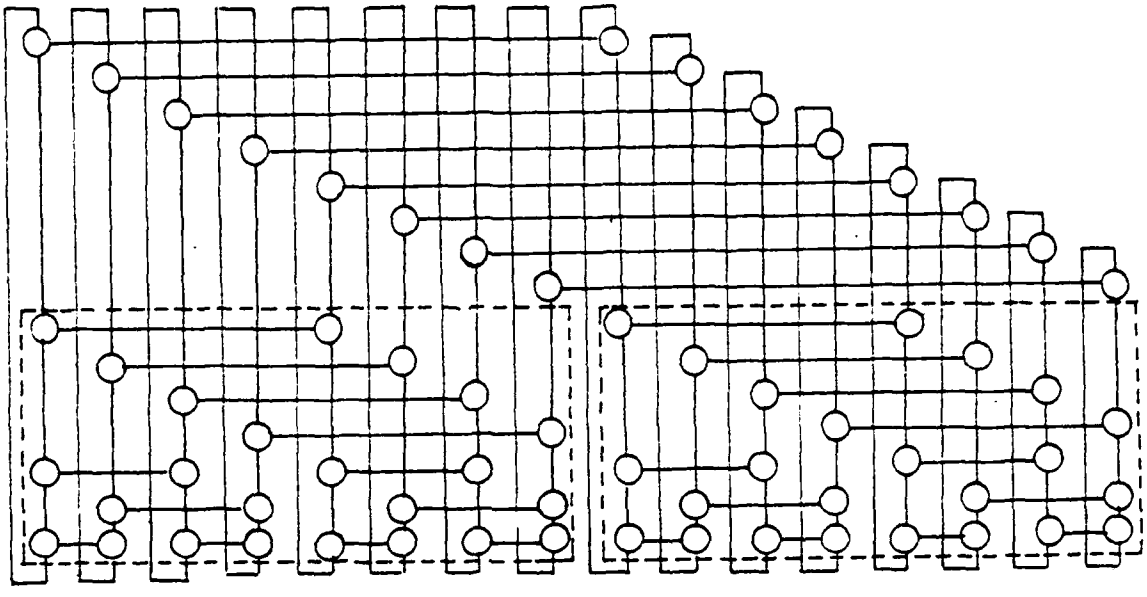


Figure 7. A standard layout for the interconnection of 4.2^4 modules.

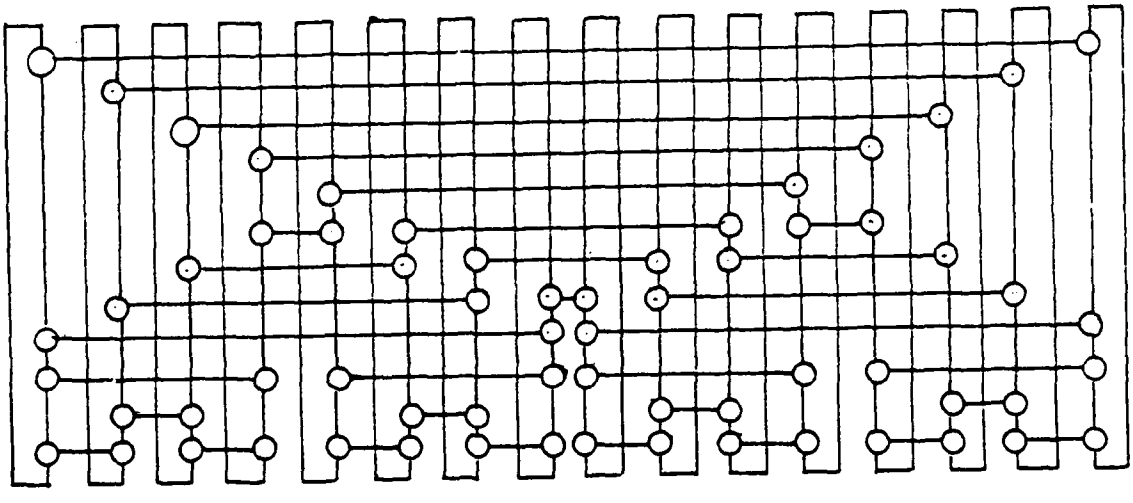


Figure 8. A more economical layout for the interconnection of 4.2^4 modules.

For pedagogical reasons, the CCC introduced so far has a number $n = s \cdot 2^s$ of processing modules with $s = 2^r$ a power of 2. A more general version of the CCC can be designed, comprising $n = h \cdot 2^s$ modules. Each of the 2^s cycles of the machine has $h \geq s$ modules. The lower $s \times 2^s$ modules of the cycles exhibit the horizontal interconnection of standard CCC, while the $(h-s) \times 2^s$ higher modules only have vertical (cycle) connections, as indicated in figure 9. Such a layout has height $2^s + h - s$ and width 2^{s+1} (in unit wire width). The programs presented in section 4 can be adapted to run on such a machine by simply ignoring operations pertaining to non-existing horizontal (external) links, and their running time is proportional to the cycle length h. We see that, for any value of h satisfying $\log_2 n \leq h \leq \sqrt{n}$, the area $\times$ $(\text{time})^2$ product

$$AT^2 = \left(\frac{n}{h} + h - \log\left(\frac{n}{h}\right)\right) \times \frac{n}{h} \times h^2 = n^2 + nh^2 - nh \log\left(\frac{n}{h}\right) = O(n^2)$$

meets the optimal theoretical bound, to within a constant factor. Of particular interest is the choice $h = O(\sqrt{n})$, which leads to a running time $T = O(\sqrt{n})$ and uses the minimal achievable area $A = O(n)$.

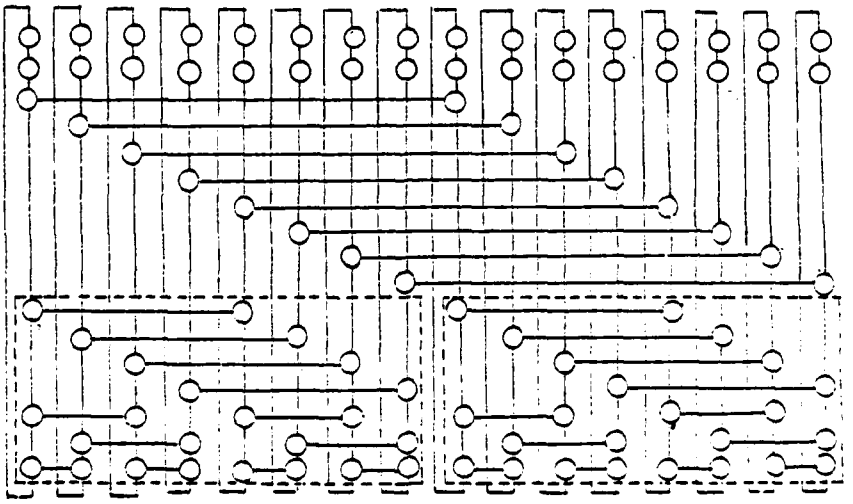


Figure 9. A standard layout for an $h \times 2^s$ CCC ($h = 6$, $s = 4$).

6. CONCLUSION

In this paper, we have proposed a structure which can be used for direct hardware implementation of specific useful algorithms, or, as suggested in section 4.3, as a general purpose parallel processing system.

We expect the CCC to be practically feasible in the present state of the technology, and to be capable of executing efficiently a wide variety of algorithms. The extent of the class of algorithms amenable to efficient CCC processing is not yet well understood, but it goes beyond the applications described in Section 1; in particular, it includes a variety of matrix and graph algorithms, as well as arithmetic and algebraic problems.

Another salient feature of this work is the possibility which appears to exist of developing a high level, general purpose language for parallel programming, which would nevertheless be automatically compilable on systems such as the CCC.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, The Analysis and Design of Computer Algorithms. Addison-Wesley, Reading, Mass., 1974.

[2] D. Heller, "A survey of parallel algorithms in numerical linear algebra," Dept. of Comp. Sci., Carnegie-Mellon University, Pittsburgh, Pa., Feb. 1976.

[3] D. S. Hirschberg, "Fast parallel sorting algorithms," Communications of the ACM, vol. 21, no. 8, pp. 657-661, 1978.

[4] L. G. Valiant, "Parallelism in comparison problems," SIAM Journal on Computing, vol. 4, 3, pp. 348-355, Sept. 1975.

[5] F. P. Preparata, "New parallel sorting schemes," IEEE Transactions on Computers, vol. C-27, no. 7, pp. 669-673, July 1978.

[6] W. M. Gentleman, "Some complexity results for matrix computation on parallel processors," Journal of the ACM, 25, 1, pp. 112-115, Jan. 1978.

[7] G. H. Barnes et al, "The ILLIAC IV computer," IEEE Transactions on Computers, vol. C-17, pp. 746-757, 1968.

[8] C. D. Thompson and H. T. Kung, "Sorting on a mesh connected computer," Proc. ACM-SIGACT Symp. on Theory of Computing, Hershey, Pa., pp. 58-64, May 1976.

[9] H. T. Kung and C. E. Leiserson, "Algorithms for VLSI processor arrays," Symposium on Sparse Matrix Computations, Knoxville, Tenn., Nov. 1978.

[10] D. Nassimi and S. Sahni, "Bitonic sort on a mesh-connected parallel computer," IEEE Transactions on Computers, vol. C-28, no. 1, pp. 2-7, Jan. 1979.

[11] L. J. Guibas, H. T. Kung and C. D. Thompson, "Direct VLSI implementation of combinatorial algorithms," Research Report, Dept. of Comp. Sci., Carnegie-Mellon University, Pittsburgh, Pa., March 1979.

[12] K. N. Levitt and W. H. Kautz, "Cellular arrays for the solution of graph problems," Comm. of the ACM, vol. 15, no. 9, pp. 789-801, 1972.

[13] M. C. Pease, "The indirect binary n-cube microprocessor array," IEEE Transactions on Computers, vol. C-26, no. 5, pp. 458-473, May 1977.

[14] A. M. Mead and L. A. Convay, Introduction to VLSI Systems. Textbook in preparation (1979).

[15] K. E. Batcher, "Sorting networks and their applications," Proc. AFIPS Spring Joint Computer Conference, vol. 32, pp. 307-314, April 1968.

[16] D. E. Knuth, The Art of Computer Programming Volume 3, Sorting and Searching. Addison-Wesley, Reading, Mass., 1973.

[17] H. S. Stone, "Parallel processing with the perfect shuffle," IEEE Transactions on Computers, vol. C-20, pp. 153-161, 1971.

[18] C. D. Thompson, "Area-time complexity for VLSI," Proc. of the 11th Annual ACM Symp. of Theory of Computing, pp. 81-88, 1979.

[19] C. D. Thompson, "A complexity theory for VLSI," Ph.D. Thesis, Carnegie-Mellon University, Dept. of Comp. Sci., 1980.

[20] M. C. Pease, "An adaptation of the Fast Fourier transform for parallel processing," Journal of the ACM, 15(2), pp. 252-264, April 1968.

[21] A Waksman, "A permutation network," Journal of the ACM, 15(1), pp. 159-163 (1968).